窗体底端

数据结构以某种形式将数据组织在一起的集合，不仅存储数据，还支持访问和处理数据的操作。

算法是为求解一个问题需要遵循的、被清楚指定的简单指令的集合。

**一、线性表**

n个数据元素的有限序列。

实现方式:

链表存储线性表的元素，即用一组任意的存储单元存储线性表的数据元素（存储单元可以是连续的，也可以是不连续的）。

* 1. **数组实现**

用一组连续的存储单元依次存储线性表的数据元素.

数组一旦创建，大小就无法改变，但当数组容量不够，可创建新的大数组来替换当前数组。

* 代码1 创建一个更大的数组来替换当前数组

|  |
| --- |
| int[] oldArray = new int[10];  int[] newArray = new int[20];  for (int i = 0; i < oldArray.length; i++) {  newArray[i] = oldArray[i];  }  // 也可以使用System.arraycopy方法来实现数组间的复制  // System.arraycopy(oldArray, 0, newArray, 0, oldArray.length);  oldArray = newArray; |

* 代码2 在数组位置index上添加元素e

|  |
| --- |
| //oldArray 表示当前存储元素的数组  //size 表示当前元素个数  public void add(int index, int e) {  if (index > size || index < 0) {  System.out.println("位置不合法...");  }  //如果数组已经满了 就扩容  if (size >= oldArray.length) {  // 扩容函数可参考代码1  }  for (int i = size - 1; i >= index; i--) {  oldArray[i + 1] = oldArray[i];  }  //将数组elementData从位置index的所有元素往后移一位  // System.arraycopy(oldArray, index, oldArray, index + 1,size - index);  oldArray[index] = e;  size++;  } |

具体可参考ArrayList集合类源码。

* 数组实现的线性表优点：可通过下标访问或修改元素，较高效
* 缺点：插入和删除花费开销较大。比如当在第一个位置前插入一个元素，首先要把所有元素往后移动一个位置。
  1. **链表**

物理存储单元上非连续、非顺序，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。

链表由一系列节点组成，节点不必在内存中相连。

节点：数据部分Data + 链部分Next。Next指向下一个节点。当添加或删除时，只需改变相关节点的Next的指向，效率很高。



单链表的结构

下面主要用代码来展示链表的一些基本操作，需要注意的是，这里主要是以单链表为例，暂时不考虑双链表和循环链表。

* + 1. 代码3 链表的节点

|  |
| --- |
| class Node<E> {  E item;  Node<E> next;  //构造函数  Node(E element) {  this.item = element;  this.next = null;  }  } |

* + 1. 代码4 定义好节点后，使用前对头节点和尾节点初始化

|  |
| --- |
| //头节点和尾节点都为空 链表为空  Node<E> head = null;  Node<E> tail = null; |

* + 1. 代码5 空链表创建一个新节点

//创建一个新节点 并让head指向此节点

head = new Node("nodedata1");

//让尾节点也指向此节点

tail = head;

* + 1. 代码6 链表追加一个节点

//创建新节点 同时和最后一个节点连接起来

tail.next = new Node("node1data2");

//尾节点指向新的节点

tail = tail.next;



* + 1. 代码7 顺序遍历链表

Node<String> current = head;

while (current != null) {

System.out.println(current.item);

current = current.next;

}

* + 1. 代码8 倒序遍历链表

|  |
| --- |
| static void printListRev(Node<String> head) {  //倒序遍历链表主要用了递归的思想(类似栈的思想)  //执行顺序: printListRev(head.next); ~ if (head != null) ~ printListRev(head.next); ~ //if (head != null) (此时不符合了，即达到边界条件，结束。开始倒序取之前放入//“栈”的函数段执行)  if (head != null)  printListRev(head.next);  System.out.println(head.item);  }  } |

* + 1. 代码 单链表反转

|  |
| --- |
| //单链表反转 主要是逐一改变两个节点间的链接关系来完成  static Node<String> revList(Node<String> head) {  if (head == null) {  return null;  }  Node<String> nodeResult = null;  Node<String> nodePre = null;  Node<String> current = head;  while (current != null) {  Node<String> nodeNext = current.next;  if (nodeNext == null) {  nodeResult = current;  }  current.next = nodePre;  nodePre = current;  current = nodeNext;  }  return nodeResult;  } |

还有像获取指定元素，移除元素等操作

链表的实现还有其它的方式:

* **循环单链表** 链表的最后一个节点指向第一个节点，整体构成一个链环。
* **双向链表** 节点中包含两个指针部分，一个指向前驱元，一个指向后继元，JDK中LinkedList集合类的实现就是双向链表。
* **循环双向链表** 最后一个节点指向第一个节点。

**二、栈与队列**

较特殊的线性表：

* 栈，访问、插入和删除元素只能在栈顶进行
* 队列，元素只能从队尾插入，从队头访问和删除。

**2.1 栈**

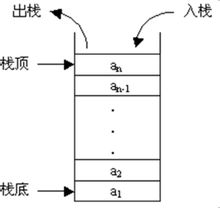
2.1.1 概述

限制插入和删除只能在一个位置(表的末端，栈顶)上进行的表。

有时又叫LIFO(Last In First Out)表，即后进先出。

push(进栈): 插入

pop(出栈): 删除最后一个元素。



2.1.2 栈的模型



答案C

因为栈也是一个表，所以任何实现表的方法都能实现栈。

|  |
| --- |
| Public class Stack<E> extends Vector<E> { |

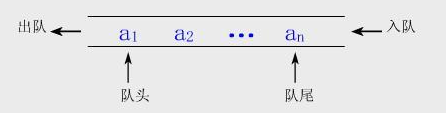
可使用LinkedList来进行栈的所有操作。

* 1. **队列**

2.2.1 概述

特殊的线性表，只许在表前端（队头 front）删除，在表后端（队尾 rear）插入

和栈一样，队列是一种操作受限制的线性表。



可用链表来实现队列

2.2.2 代码9 简单实现队列类（利用LinkedList）

|  |
| --- |
| public class MyQueue<E> {  private LinkedList<E> list = new LinkedList<>();  // 入队  public void enqueue(E e) {  list.addLast(e);  }  // 出队  public E dequeue() {  return list.removeFirst();  }  } |

* 普通队列: 先进先出
* 优先队列: 元素被赋予优先级。访问元素时，具有最高优先级的元素最先被删除。

Java集合框架中，类PriorityQueue是优先队列的实现类

**三、树与二叉树**

重要的非线性数据结构，以树和二叉树最为常用。

**3.1 树**

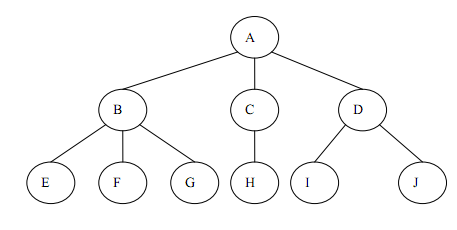
**树** 是由n（n>=1）个有限节点组成一个具有层次关系的集合。

特点：每个节点有零个或多个子节点；

**根**节点: 没有父节点的节点

每一个非根节点有且只有一个**父节点** ；

除了根节点外，每个子节点可以分为多个不相交的子树。



**3.2 二叉树**

**3.2.1 定义**

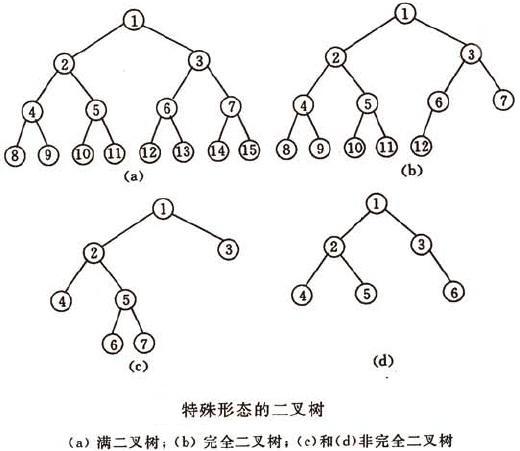
二叉树:每个节点最多有两棵子树的树结构。

子树被称作“左子树”和“右子树”。

常被用于实现二叉查找树和二叉堆。

**3.2.2 相关性质**

* 每个结点至多2棵子树
* 子树有左右之分，次序不能颠倒
* 第i层至多有2^(i-1)个结点；(2^0 2^1 2^2…)
* 深度为k的二叉树至多有2^k-1个结点.(2^0 + 2^1 + 2^2 + …+2^(k-1)= 2^k -1)
* 深度为k，且有2^k-1个节点的二叉树称为 **满二叉树** ；
* 深度为k，有n个节点的二叉树，当且仅当其每一个节点都与深度为k的满二叉树中，序号为1至n的节点对应时，称之为 **完全二叉树** 。



**3.2.3 三种遍历方法**

在树中查找具有某种特征的节点，或对树中全部节点进行某种处理，就涉及到二叉树的遍历。

二叉树主要由3个基本单元组成，根节点、左子树和右子树。

如限定先左后右，根据这三个部分遍历顺序不同，可分为先序、中序和后续遍历。

* + - 1. **先序遍历** (根-左-右)

若二叉树为空，则空操作

否则先访问根节点，再先序遍历左子树，最后先序遍历右子树。

* + - 1. **中序遍历 （左-根-右）**

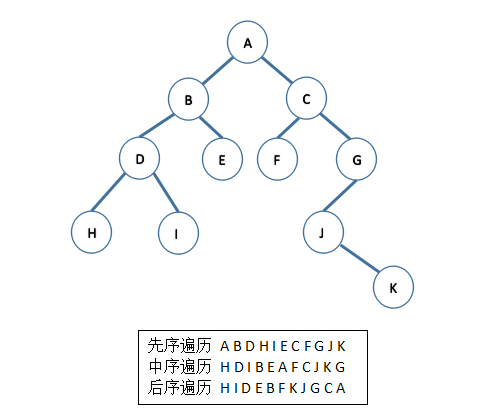
若二叉树为空，则空操作

否则先中序遍历左子树，再访问根节点，最后中序遍历右子树。

* + - 1. **后序遍历 （左右根）**

若二叉树为空，则空操作

否则先后序遍历左子树访问根节点，再后序遍历右子树，最后访问根节点。



**3.3 树和二叉树的区别**

(1) 二叉树每个节点最多有2个子节点，树则无限制。

(2) 二叉树中节点的子树分为左子树和右子树，即使某节点只有一棵子树，也要指明该子树是左子树还是右子树，即二叉树是有序的。

(3) 树决不能为空，至少有一个节点；而二叉树可为空。

**3.4 二叉查找树**

**3.4.1 定义**

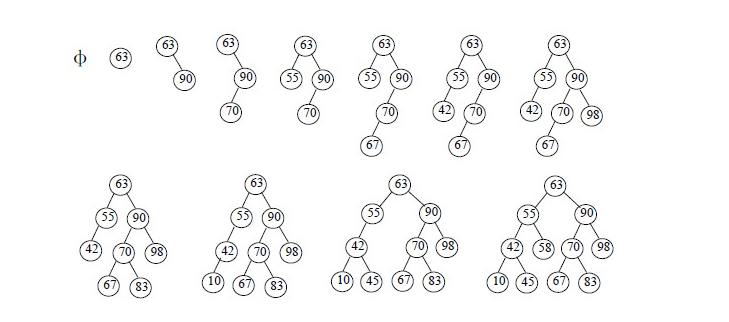
* 就是二叉排序树，也叫二叉搜索树。
* 二叉查找树或为空树，或是具有下列性质的二叉树：

(1) 若左子树不空，则左子树上所有结点的值均小于它的根结点的值；（左<根）

(2) 若右子树不空，则右子树上所有结点的值均大于它的根结点的值；（根<右）

(3) 左、右子树也分别为二叉排序树；

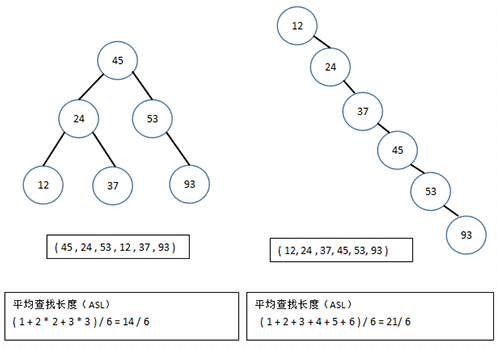
(4) 没有键值相等的结点。



典型的二叉查找树的构建过程

**3.4.2性能分析**

对二叉查找树来说，当给定值相同但顺序不同，所构建的二叉查找树形态不同



不同形态, 平衡二叉树的ASL不同

含有n个节点的二叉查找树的平均查找长度和树的形态有关。

最坏情况下，当先后插入的关键字有序时，构成的二叉查找树蜕变为单支树，树的深度为n，其平均查找长度(n+1)/2(和顺序查找相同），

最好的情况是二叉查找树的形态和折半查找的判定树相同，其平均查找长度和log2(n)成正比。

平均情况下，二叉查找树的平均查找长度和logn是等数量级的，所以为了获得更好的性能，通常在二叉查找树的构建过程需要进行“平衡化处理”，之后我们将介绍平衡二叉树和红黑树，这些均可以使查找树的高度为O(log(n))。

**3.4.3二叉树的节点先序/中序/后序遍历 代码**

二叉查找树的三种遍历都可以直接用递归的方法来实现：

* TreeNode.java

|  |
| --- |
| package tree;  public class TreeNode<E> {    E element;  public TreeNode<E> left;  public TreeNode<E> right;    public TreeNode(E element) {  this.element = element;  }    //先序遍历  public void preorder(TreeNode<E> root) {  if(root == null) {  return;  }    System.out.println(root.element + "");    preorder(root.left);  preorder(root.right);    }    //中序遍历  public void inorder(TreeNode<E> root) {  if(root == null) {  return;  }    inorder(root.left);    System.out.println(root.element + "");    inorder(root.right);  }    //代码14 后序遍历  public void postorder(TreeNode<E> root) {  if (root == null) {  return;  }  postorder(root.left);  postorder(root.right);  System.out.println(root.element + " ");  }  } |

测试代码：

|  |
| --- |
| package test;  import org.junit.Test;  import tree.TreeNode;  public class TestTreeNode {  @Test  public void test() {  TreeNode treeNode = new TreeNode("rootdata1");  treeNode.left = new TreeNode("leftdata1");  treeNode.right = new TreeNode("rightdata1");    treeNode.preorder(treeNode);  //treeNode.inorder(treeNode);  //treeNode.postorder(treeNode);  }  } |

**3.4.4 二叉查找树的简单实现**

|  |
| --- |
| package tree.binsearchtree;  public class MyBinSearchTree<E extends Comparable<E>> {    //根  private TreeNode<E> root;    // 默认构造函数  public MyBinSearchTree(TreeNode treeNode) {  this.root = treeNode;  }    // 二叉查找树的搜索  public boolean search(E element) {    TreeNode<E> current = root;  while (current != null) {  if(element.compareTo(current.element) < 0) {  current = current.left;  } else if (element.compareTo(current.element) > 0) {  current = current.right;  } else {  return true;  }  }  return false;  }    // 二叉查找树的插入  public boolean insert(E element) {  // 如果之前是空二叉树 插入的元素就作为根节点  if(root == null) {  root = createNewNode(element);  } else {  // 否则就从根节点开始遍历 直到找到合适的父节点  TreeNode<E> parent = null;  TreeNode<E> current = root;    while (current != null) {  if(element.compareTo(current.element) < 0) {  parent = current;  current = current.left;  //要插入的元素(5)比父节点(20)小，将其与该父节点的左节点(10)比较 ;发现比10还要小，继续想与10的左节点比较  //而10没有左节点，跳出循环 将该元素与10比较，小——创建节点  } else if(element.compareTo(current.element) > 0) {  parent = current;  current = current.right;  } else {  return false;  }  }    //插入  if(element.compareTo(parent.element) < 0) {  parent.left = createNewNode(element);  } else {  parent.right = createNewNode(element);  }    System.out.println(parent.toString());  }    return true;  }    // 创建新的节点  protected TreeNode<E> createNewNode(E e) {  return new TreeNode(e);  }  } |

上面代码展示了一个自己实现的简单的二叉查找树，包括了几个常见的操作。

因在二叉查找树中删除节点的操作比较复杂，下面详细介绍一下。

s

**3.4.4.2二叉查找树中删除节点分析**

要在二叉查找树中删除一个元素，首先需定位包含该元素的节点，及它的父节点。假设current指向二叉查找树中包含该元素的节点，而parent指向current节点的父节点，current节点可能是parent节点的左孩子或右孩子。

需考虑两种情况：

|  |
| --- |
| * 1.current节点没左孩子，只需将patent节点和current节点的右孩子相连 * 2. current节点有一个左孩子，假设rightMost指向包含current节点的左子树中最大元素的节点，而parentOfRightMost指向rightMost节点的父节点。那么先使用rightMost节点中的元素值替换current节点中的元素值，将parentOfRightMost节点和rightMost节点的左孩子相连，然后删除rightMost节点。   对2的总结：也就是从current节点的左子树中中找出最大的节点（该节点一定没有右孩子），替换current节点。 |

|  |
| --- |
| // 二叉搜索树删除节点  public boolean delete(E e) {  TreeNode<E> parent = null;  TreeNode<E> current = root;  // 找到要删除的节点的位置  while (current != null) {  if (e.compareTo(current.element) < 0) {  parent = current;  current = current.left;  } else if (e.compareTo(current.element) > 0) {  parent = current;  current = current.right;  } else {  break;  }  }  // 没找到要删除的节点  if (current == null) {  return false;  }  // 考虑第一种情况  if (current.left == null) {  if (parent == null) {  root = current.right;  } else {  if (e.compareTo(parent.element) < 0) {  parent.left = current.right;  } else {  parent.right = current.right;  }  }  } else { // 考虑第二种情况  TreeNode<E> parentOfRightMost = current;  TreeNode<E> rightMost = current.left;  // 找到左子树中最大的元素节点（该节点无右节点）  while (rightMost.right != null) {  parentOfRightMost = rightMost;  rightMost = rightMost.right;  }  // 替换  current.element = rightMost.element;  // parentOfRightMost和rightMost左孩子相连  if (parentOfRightMost.right == rightMost) {  parentOfRightMost.right = rightMost.left;  } else {  parentOfRightMost.left = rightMost.left;  }  }  return true;  } |

**3.5 平衡二叉树**

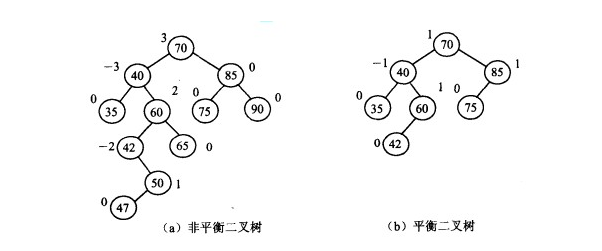
3.5.1 概述

又称AVL树

或是空树

或是具有下列性质的二叉树：

左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差绝对值不超过1。



3.5.2平衡二叉树

AVL树是最先发明的自平衡二叉查找树算法。

在AVL中任何节点的两个儿子子树的高度最大差别为1，也被称为高度平衡树，n个结点的AVL树最大深度约1.44log2n。

查找、插入和删除在平均和最坏情况下都是O（log n）。

增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。

3.5.3红黑树

平衡二叉树的一种，保证最坏情况下基本动态集合操作的事件复杂度为O(log n)。

红黑树和平衡二叉树区别：

(1) 红黑树放弃了追求完全平衡，追求大致平衡，在与平衡二叉树的时间复杂度相差不大的情况下，保证每次插入最多只需要三次旋转就能达到平衡，实现起来也更为简单。

(2) 平衡二叉树追求绝对平衡，条件比较苛刻，实现起来比较麻烦，每次插入新节点之后需要旋转的次数不能预知。

**四、图**

* **4.1 简介**

图是一种较线性表和树更为复杂的数据结构，在线性表中，数据元素之间仅有线性关系，在树形结构中，数据元素之间有着明显的层次关系，而在图形结构中，节点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关。图的应用相当广泛，特别是近年来的迅速发展，已经渗入到诸如语言学、逻辑学、物理、化学、电讯工程、计算机科学以及数学的其他分支中。